PROGRAMANDO EN HASKELL

Capítulo 6 - Funciones Recursivas

Introducción

Como ya hemos visto, muchas funciones pueden ser definidas de manera natural en términos de otras funciones.

```
fac :: Int → Int
fac n = product [1..n]
```

fac mapea cualquier entero n al producto de enteros entre 1 y n.

Las expresiones son evaluadas por un proceso paso a paso de aplicación de funciones a sus argumentos.

Por ejemplo:

```
    fac 4
=
    product [1..4]
=
    product [1,2,3,4]
=
    1*2*3*4
=
    24
```

Funciones Recursivas

En Haskell, las funciones también se pueden definir en términos de sí mismas. Tales funciones se llaman recursivas.

```
fac 0 = 1
fac n = n * fac (n-1)
```

fac mapea 0 a 1, y cualquier otro entero al producto de sí mismo y al factorial de su predesor.

Por ejemplo:

```
  fac 3
=
  3 * fac 2
=
  3 * (2 * fac 1)
=
  3 * (2 * (1 * fac 0))
=
  3 * (2 * (1 * 1))
=
  3 * (2 * 1)
=
  3 * 2
=
  6
```

Nota:

- fac 0 = 1 es apropiado porque 1 es la identidad para la multiplicación: 1*x = x = x*1.

- La definición recursiva diverge sobre enteros < 0 porque el caso base nunca se cumple:

```
> fac (-1)

*** Exception: stack overflow
```

¿Por qué es útil la Recursión?

- Algunas funciones, como factorial, son mas simples de definir en términos de otras funciones.

- Sin embargo, como ya veremos, muchas funciones pueden definirse naturalmente en términos de sí mismas.

- Las propiedades de las funciones definidas usando recursión pueden ser probadas con la técnica matemática simple pero poderosa de inducción.

Recursión sobre Listas

La recursión no está restringida a los números, sino que también puede usarse para definir funciones sobre listas.

```
product :: Num a ⇒ [a] → a
product []     = 1
product (n:ns) = n * product ns
```

product mapea la lista vacía a 1, y cualquier lista no vacía a su cabezal multiplicado por el producto de sus elementos restantes.

Por ejemplo:

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * (3 * product [4])
=
  2 * (3 * (4 * product []))
=
  2 * (3 * (4 * 1))
=
  24
```

Usando el mismo patrón de recursión como en product podemos definir la función length sobre listas.

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

length mapea la lista vacía a 0, y cualquier lista no vacía al sucesor del tamaño de sus elementos restantes.

Por ejemplo:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

Usando un patrón de recursión similar podemos definir la función reverse sobre listas.

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse mapea la lista vacía a la lista vacía, y cualquier lista no vacía a sus elementos restantes invertidos añadidos a su cabezal.

Por ejemplo:

```
  reverse [1,2,3]
=
  reverse [2,3] ++ [1]
=
  (reverse [3] ++ [2]) ++ [1]
=
  ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Múltiples Argumentos

Las funciones con mas de un argumento también pueden definirse usando recursión. Por Ejemplo:

- zip sobre los elementos de dos listas:

```
zip :: [a] → [b] → [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- Borrar los primeros n elementos de una lista:

```
drop :: Int → [a] → [a]
drop 0 xs      = xs
drop _ []      = []
drop n (_:xs) = drop (n-1) xs
```

- Añadiendo dos listas:

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Quicksort

El algoritmo quicksort para ordernar una lista de valores se puede especificar con las siguientes dos reglas:

- La lista vacía ya está ordenada;
- Las listas no vacías se pueden ordenar al ordenar los elementos restantes $\leq$ al cabezal, ordenando los elementos restantes $>$ que el cabezal, añadiendo las listas resultantes en cada lado del valor cabezal.

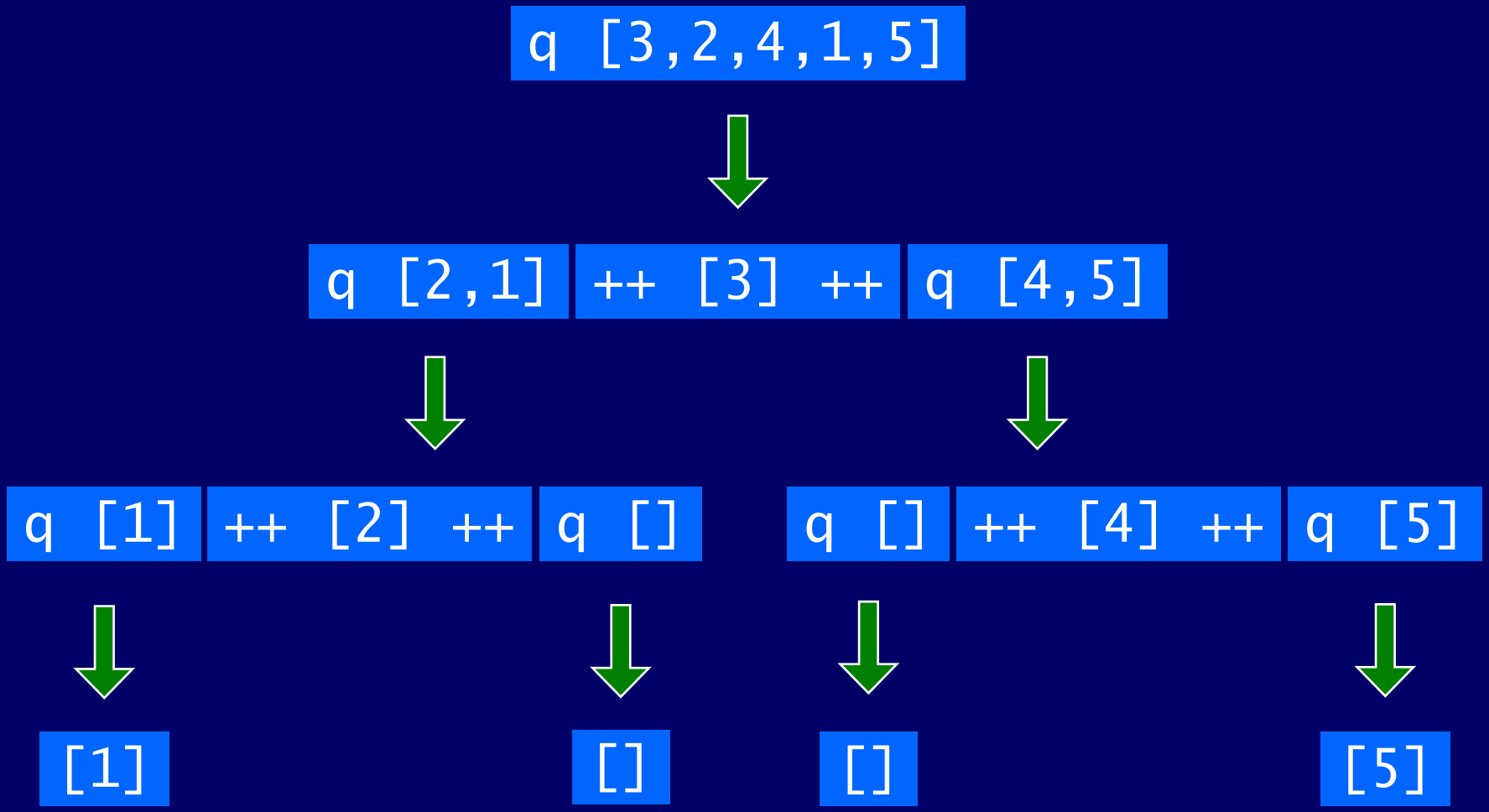
Con recursión, dicha especificación se traduce directamente a su implementación:

```
qsort :: Ord a ⇒ [a] → [a]
qsort []     = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

Nota:

- ¡Probablemente esta es la implementación mas simple de quicksort en cualquier lenguaje de programación!

Por ejemplo (abreviando qsort como q):



Ejercicios

(1) Sin ver el código fuente de la librería estándar, define las siguientes funciones presentes en la librería usando recursión:

- Decidir si todos los valores lógicos en una lista son verdaderos:

```
and :: [Bool] → Bool
```

- Concatena una lista de listas:

```
concat :: [[a]] → [a]
```

- Producir una lista con elementos idénticos:

```
replicate :: Int → a → [a]
```

- Seleccionar el n-avo elemento de una lista:

```
(!!) :: [a] → Int → a
```

- Decidir si un valor es un elemento de una lista:

```
elem :: Eq a ⇒ a → [a] → Bool
```

(2) Define una función recursiva

```
merge :: Ord a ⇒ [a] → [a] → [a]
```

que mezcla dos listas ordenadas de valores para brindar una sola lista ordenada.
Ejemplo:

```
> merge [2,5,6] [1,3,4]

[1,2,3,4,5,6]
```

(3) Define una función recursiva

```
msort :: Ord a ⇒ [a] → [a]
```

que implementa el algoritmo merge sort, puede ser especificado con las siguientes dos reglas:

- Listas de tamaño ≤ 1 ya están ordenadas;
- Las otras listas se ordenan al ordenar por mitades y mezclar las listas resultantes.

Fuente.

Profesor Graham Hutton.
http://www.cs.nott.ac.uk/~pszgmh/

Traducción al español.

Caleb Josue Ruiz Torres.
https://www.calebjosue.com